# CSML: high-level bindings between C# and OCaml

Alain Frisch
LexiFi SAS

August 2008

# Contents


**1 Foreword** **5**
- 1.1 License and availability . . . 5
- 1.2 Requirements . . . 5

**2 Design principles and goals** **7**

**3 CSML scripts** **9**
- 3.1 Overview of the system . . . 9
- 3.2 Basic types . . . 10
  - 3.2.1 Integers . . . 10
  - 3.2.2 Strings . . . 11
  - 3.2.3 Blobs . . . 11
  - 3.2.4 `void/unit` . . . 11
- 3.3 Structural types . . . 11
  - 3.3.1 List and arrays . . . 11
  - 3.3.2 Options, nullable types . . . 12
  - 3.3.3 Tuples . . . 12
  - 3.3.4 Functions . . . 13
- 3.4 Organizing the imported components . . . 14
  - 3.4.1 Choosing file names . . . 14
  - 3.4.2 Namespaces and classes in C# . . . 15
  - 3.4.3 Modules in OCaml . . . 16
  - 3.4.4 Inline code . . . 16
- 3.5 Exceptions . . . 17
- 3.6 Exporting OCaml values opaquely . . . 18
  - 3.6.1 Running example . . . 18
  - 3.6.2 Extending the class with custom code . . . 20
  - 3.6.3 About garbage collection . . . 20
- 3.7 Exporting C# values opaquely . . . 21
  - 3.7.1 Running example . . . 21
  - 3.7.2 Special methods . . . 22
  - 3.7.3 Indexed accesors . . . 23
  - 3.7.4 Accessing sub-components . . . 23
  - 3.7.5 Ignoring the result . . . 24

3.7.6 Weak references . . . . . 24
3.8 Binding C# classes to OCaml classes . . . . . 24
3.8.1 Inheritance . . . . . 25
3.9 Structurally reflecting OCaml datastructures into C# classes . . 26
3.9.1 Records . . . . . 26
3.9.2 Variants . . . . . 28
3.10 Mapping C# enumerations into OCaml variants of polymorphic variants . . . . . 30
3.11 Resolution of recursive types . . . . . 31
3.12 Dependent CSML scripts . . . . . 33

**4 Using CSML** **35**
4.1 Using the compiler . . . . . 35
4.2 Computing dependencies . . . . . 35
4.3 Linking . . . . . 35
4.3.1 Initialization . . . . . 36
4.3.2 Static linking . . . . . 36
4.3.3 Dynamic linking . . . . . 36
4.3.4 Linking the OCaml code without the C# part . . . . . 37
4.4 A note on initialization order . . . . . 37
4.5 A note on environment variables . . . . . 37

**5 Formal syntax** **39**

# Chapter 1

# Foreword

This manual documents the CSML system. It is organized as follows.

- Chapter 2 explains some design principles and goals of CSML.
- Chapter 3 gives an overview of the system and of CSML scripts.
- Chapter 4 explains how to use CSML in practice.
- Chapter 5 gives the formal syntax of CSML scripts.

## 1.1 License and availability

The home page for CSML system is `http://www.lexifi.com/csml`.

The CSML compiler is distributed free of charge, but only in binary form. The CSML runtime support is made of C, OCaml and C# units. It is distributed in source form under the terms of an open-source license (LGPL + the typical linking exception for OCaml libraries). The details of the licenses for the CSML compiler and the CSML runtime support are included in the LICENSE file of the distribution.

LexiFi SAS is willing to cooperate with other people interested in improving the CSML system. Feel free to apply for a source distribution of the compiler.

## 1.2 Requirements

See the README file in the distribution.

# Chapter 2

# Design principles and goals

CSML is a system to help developing applications that freely mix OCaml and C# code.

CSML has been developed by LexiFi SAS as an internal development tool. LexiFi has decided to distribute it to help people write their Windows applications with OCaml. LexiFi uses CSML for the following scenarios:

- Wrapping .Net libraries (like Winforms) to make them available to OCaml applications.
- Providing a .Net API to OCaml applications.
- Developping mixed applications, where some parts are implemented in OCaml and other parts in C#.

Technically, both OCaml and C# provide relatively low-level interfaces to C. CSML relies on these internally but it exposes much higher-level concepts. In particular, some care has been taken to ensure the following properties.

**High-level binding** CSML hides to the user the complexity of low-level bindings such as memory management, in-memory layout of objects, translation of complex values.

**User-control** The user is in charge of organizing the binding code as he sees fit. Basically, one describes the interface that one expects to get in one language when importing components from the other language. In particular, it is possible to choose custom names for imported components and to organize them into classes or namespaces in C#, and classes or modules in OCaml.

**Type-safety** The interface does not compromise static type-safety as provided by each language. Type information from each language is reflected in the other and is checked at compile-time.

**Opaque or structural translation of values** The system supports passing value abstractly from one language to the other (which means that it is seen as an opaque value there); it also supports translating datastructures in a structural way (basic types, sum types, record types, lists, arrays, options, ...).

**GC-safety** The user of CSML does not need to worry about making the two garbage collectors live together. CSML takes care of registering and releasing GC pointers from one heap to the other one. That said, the system also offers some fine-tuning features to optimize performance and deal with tricky cases like circular references between the two heaps.

**Support for exceptions** Exceptions raised in one language are wrapped in the other language as regular exceptions, and round-tripping is supported (e.g. a C# exception that escapes to OCaml and then back to C# will retain all the information from the original one).

**Support for first-class functions** First-class functions (in OCaml) and delegates (in C#) are considered seriously in CSML. The system relies on predefined C# generic delegates to mimic the structural aspect of OCaml arrow types. A C# delegate is wrapped into an OCaml closure when it enters to OCaml world. Similarly, an OCaml closure is wrapped into a C# delegate when it goes to the C# world; if this delegate is later sent back to OCaml, it is unwrapped to retrieve the original closure instead of being wrapped again.

# Chapter 3

# CSML scripts

## 3.1 Overview of the system

The CSML system is made of a compiler and a runtime support library. The compiler takes a CSML script which describes C# and OCaml components to be exported from their native language to the other one, and produces a number of OCaml and C# source files. Those files have to be compiled and linked with the rest of the application and the CSML runtime support library.

The CSML scripts usually have the extension `.csml`. They describe how imported components should look like in their target language. For instance, here is one section of such a script:

```
mlfile "getting_started_cs.ml"

module FooBar: sig
  val bipbip: int -> int -> unit = static Getting.Started.Foo.BipBip
end
```

This section tells the compiler to produce an OCaml file `getting_started_cs.ml` that imports the static method `Getting.Started.Foo.BipBip` defined in C#. This method should be wrapped as an OCaml function called `bipbip` located in a submodule `FooBar` of the OCaml compilation unit `Getting_started_cs`. To be compatible with the code above, the C# method must have the following signature:

```
public static void BipBip(int, int);
```

If the signature of the method is different, there is a type-error when the C# files generated by the script are compiled.

Let us look at an example of how to import functions from OCaml to C#.

```
csfile "getting_started_csml.cs"

namespace Getting.Started {
  public static class Foo {
    public static class MyStaticClass {
      public static string DoSomething(string) = Getting_started.do_something;
    }
  }
}
```

This section tells the compiler to produce a C# file `getting_started_csml.cs` which defines a static class `Foo` in the namespace `Getting.Started`, with a static inner-class `MyStaticClass` that contains a single static method `DoSomething`. This method is imported from the OCaml function `Getting_started.do_something`, which must have type `string -> string`.

Here are some more examples of OCaml signatures together with their C# equivalent (for CSML):

| OCaml | C# |
|---|---|
| `unit -> unit` | `void F();` |
| `int -> unit` | `void F(int);` |
| `unit -> int` | `int F();` |
| `int -> int -> unit` | `void F(int, int);` |
| `int -> int -> int` | `int F(int, int);` |

## 3.2 Basic types

For most basic types, there is a natural correspondence between C# and OCaml.

| C# | OCaml |
|---|---|
| `void` | `unit` |
| `int` | `int` |
| `long` | `int64` |
| `double` | `float` |
| `string` | `string` |
| `bool` | `bool` |
| `Exception` | `exn` |

### 3.2.1 Integers

The OCaml type `int` stores 31-bit integers (on 32-bits machines). When a C# integer that does fit within 31-bit is translated to OCaml, the behavior is not specified.

### 3.2.2 Strings

C# strings represent sequences of Unicode characters. When copied to and from C#, OCaml strings are interpreted as sequences of characters in the Latin1 (iso-9959-1) subset of Unicode. If a C# string that contains code points >= 256 is copied to OCaml, the result is undefined. An OCaml string copied to C# and then back to OCaml is guaranteed to be equal to the original string.

### 3.2.3 Blobs

OCaml strings are often used as sequences of bytes. To support this common case, CSML understands a special type, written `byte[]` in C# signatures and `blob` in OCaml signatures. This type is mapped to the OCaml type `string` and to the C# type `byte[]`.

For instance, the following specification assumes that the imported OCaml function has type `string -> string`:

```
public static string f(byte[]) = M.f;
```

and the following specification assumes that the imported C# method has signature `byte[] F(byte[])`.

```
val f: blob -> blob = static C.F
```

### 3.2.4 `void/unit`

The C# `void` "type" can only appear as the result type of a method. It corresponds to the result type `unit` in OCaml. A C# method that takes an empty list of arguments corresponds to an OCaml function that takes a single argument of type `unit`. The type `unit` can only appear as the single argument of a function and/or as the result of a function.

## 3.3 Structural types

Structural types (lists, arrays, options, tuples) are very common in OCaml signatures.

### 3.3.1 List and arrays

List and arrays have a natural correspondence in C#.

| C# | OCaml |
|---|---|
| T[] | 'a array |
| List<T> | 'a list |

The C# generic class `List` is defined in the namespace `System.Collections.Generic` (but one simply write `List` in CSML scripts). The following line is an example of how to import in OCaml a C# static method of signature `List<string[]> F(List<int>)`.

```
val f: int list -> string array list = static C.F
```

### 3.3.2 Options, nullable types

In OCaml, the built-in parameterized type `option` is defined as:

```
type 'a option = Some of 'a | None
```

CSML's standard library defines a C# generic class `Lexifi.Interop.Option<T>` to reflect this type. Here is the current signature for this class:

```
public class OptionNoneException : Lexifi.LexiFi_exception {
  public OptionNoneException(Type T);
}
public class Option<T> {
  public Option(T x);  // Create a Some option
  public Option();     // Create a None option
  public bool Is_some{ get; } // Check whether the option is Some
  public void Clear(); // Set the option to None
  public T Val { get; set; } // Get or set the value
}
```

The getter of the property `Val` throw an exception `Lexifi.Interop.OptionNoneException` if the current value of the option is `None`.

In some cases, we do not want to reflect an OCaml option type explicitly in C#. Instead, we might like to use the C# `null` value to represent `None`. The CSML compiler understands a pseudo-type written $\alpha$ `nullable` in OCaml parts and `Nullable<T>` in C# parts. This type really means $\alpha$ `option` in OCaml and just `T` in C#: `Some` values are mapped to and from values of type `T` and `None` is mapped to and from `null`. Here is an example that shows how to use this feature to import a function from C# to OCaml:

```
val f: unit -> string nullable = static Myclass.f
```

The resulting OCaml function `f` has type `unit -> string option`, but the static method `Myclass.f` returns a `string`, not a `Option<string>` as would be the case if `option` had been used instead of `nullable`.

### 3.3.3 Tuples

| C# | OCaml |
|---|---|
| Tuple<T1,...Tn> | 'a1 * ... * 'an |

To reflect OCaml tuple types, CSML's standard library defines C# generic classes `Lexifi.Interop.Tuple<T0,T1>`, `Lexifi.Interop.Tuple<T0,T1,T2>`, ..., `Lexifi.Interop.Tuple<T0,T1,...,T10>`. Here is the current signature for one of them:

```
  public class Tuple<T0, T1, T2, T3> {
    public Tuple(T0 arg0, T1 arg1, T2 arg2, T3 arg3);
    public T0 TVal0 { get; set }
    public T1 TVal1 { get; set }
    public T2 TVal2 { get; set }
    public T3 TVal3 { get; set }
  }
```

Let us look at the following example of a CSML script fragment:

```
public static class C {
  public static Tuple<Option<int>,string,int>
                    f(Option<Tuple<string,string>>) = M.f;
}
```

The imported OCaml function `M.f` must have type `(string * string) option -> (int option * string * int)`. Here is an example of how to call the imported function:

```
  Tuple<Option<int>,string,int> r =
    C.f(new Option<Tuple<string,string>>(
           new Tuple<string,string>("a", "b")));
  if (r.TVal0.Is_some)
    System.Console.WriteLine("I = " + r.TVal0.Val);
  System.Console.WriteLine("S = " + r.TVal1);
```

### 3.3.4 Functions

| C# | OCaml |
|---|---|
| ArrowVoid | unit -> unit |
| Arrow<S> | unit -> 'a |
| ArrowVoid<T1,...Tn> | 'a1 -> ... -> 'an -> unit |
| Arrow<T1,...Tn,S> | 'a1 -> ... -> 'an -> 'b |

Function types and first-class functions are key features of the OCaml language. They are also available in C#. CSML's standard library defines families of C# generic delegates as the counterpart of OCaml's function types:

```
  public delegate S Arrow<S>();
  public delegate S Arrow<T0, S>(T0 arg0);
  public delegate S Arrow<T0, T1, S>(T0 arg0, T1 arg1);
  public delegate S Arrow<T0, T1, T2, S>(T0 arg0, T1 arg1, T2 arg2);
...
```

```
public delegate void ArrowVoid();
public delegate void ArrowVoid<T0>(T0 arg0);
public delegate void ArrowVoid<T0, T1>(T0 arg0, T1 arg1);
public delegate void ArrowVoid<T0, T1, T2>(T0 arg0, T1 arg1, T2 arg2);
```

The `Ti` correspond to the types of the arguments and `S` corresponds to the type of the result.

For instance, the OCaml types `int -> string -> string` and `int -> string -> unit` are mapped respectively to the C# types `Arrow<int,string,string>` and `ArrowVoid<int,string>`.

Note that in OCaml, the two types `int -> string -> string` and `int -> (string -> string)` are strictly equivalent. For CSML, however, they are mapped to two different types, namely `Arrow<int,string,string>` and `Arrow<int,Arrow<string,string>>`.

OCaml functional values are thus mapped to and from C# delegates. In practice, a small wrapper is added to translate the arguments and result, and to call the original function or delegate. When an OCaml function is translated to a C# delegate and the same delegate is sent again to OCaml, the original value is used (unwrapped), instead of a re-wrapped version of the delegate (as would be the case for a delegate that did not originate from an OCaml function). This avoids the problematic situation where a function which is sent back and forth between the two languages would become fatter each time because of the layers used while wrapping a function as a delegate or a delegate as a function.

## 3.4 Organizing the imported components

CSML lets the programmer choose how the imported components should be organized in the target language.

### 3.4.1 Choosing file names

A CSML script is made of several sections. Each one describes the content of a single source file (OCaml or C#) to be produced by the compiler. For instance, the following script will produce three OCaml files and three C# files.

```
mlstub "ml_stub.ml"
csstub "cs_stub.cs" InitClass

mlfile "mymodule1.ml"
...

mlfile "mymodule2.ml"
...

csfile "myfile1.cs"
```

```
...

csfile "myfile2.cs"
...
```

The directive `mlstub` lets us choose the name of a special OCaml source file that contains all the code needed to export OCaml values to C#. This module should be linked after all the OCaml modules that define values to be exported to C#.

Similarly, the directive `csstub` allows us to choose the name of a special C# source file that contains all the code to export C# components to OCaml and all the glue code between C# and OCaml. The directive also takes a second argument. It is the name of a class to be created in the namespace `LexiFi.Interop`. This class has a static method `void Init()` that must be called (it forces C# components to be actually exported to OCaml). In the example above, the call should look like `Lexifi.Interop.InitClass.Init();`.

It is possible to reuse the same name as for the stub files in normal `mlfile` or `csfile` sections; in that case, the stub code will simply be appended to the corresponding file.

The decision to split the C# part of the script into several files makes it possible to link the resulting files into different .Net assemblies.

This ability to produce several files is much more important for the OCaml side. Indeed, the name of the `.ml` files constitutes the first layer in the module hierarchy for fully qualified OCaml names. Also, because OCaml compilation units cannot be mutually recursive, it is sometimes necessary to split the CSML script. For instance, one could imagine in the example above that `mymodule1.ml` imports some components from C# that are used in a module `foobar.ml` (written by hand) and that `mymodule2.ml` refers to types defined in `foobar.ml`. In such a case, the code for `mymodule1.ml` and `mymodule2.ml` cannot be merged.

### 3.4.2 Namespaces and classes in C#

The C# parts of CSML scripts can be organized in several namespaces and classes. Namespaces can be defined at the toplevel or within another namespace. Classes can be defined at the toplevel or within another class or namespace. The following modifiers are recognized for classes: `public`, `private`, `static`. The classes generated by the CSML compiler are always declared `partial`, which makes it possible to extend them in hand-written C# source files (as long as they are compiled and linked together with the files generated by the CSML compiler).

```
csfile "myfile1.cs"

namespace Foo {
  namespace Bar {
```

```
    public class A {
      public static class B {
        public static int F1(int) = M.f1;
      }
    }
    public class C {
      public static int X { get = M.f2; set = M.f3; }
    }
  }
}
```

In this example we import three functions from OCaml. The function `M.f1` must have type `int -> int`; it is wrapped as a static method in the class `Foo.Bar.A.B`. The function `M.f2` is wrapped as the getter for the static property `Foo.Bar.C.X`; it must have type `unit -> int`. Similarly, the function `M.f3` is wrapped as the setter for the same property. It is possible to define properties with only a getter or a setter.

### 3.4.3 Modules in OCaml

The OCaml parts of CSML scripts can be organized in (nested) modules.

```
mlfile "mymodule1.ml"

module M1: sig
  module A: sig
    val f: int -> int = static MyClass.F1
  end
end

val f: int -> int -> unit = static MyClass.F2

val g: unit -> string = static get MyClass.G
val h: string -> unit = static set MyClass.G
```

In this example we import two static methods from the C# class `MyClass` as two OCaml function `Mymodule1.M1.A.f` and `Mymodule1.f`. In addition, the OCaml functions `Mymodule1.g` and `Mymodule1.h` import the getter and setter for the static method `MyClass.G`.

It is also possible to use OCaml classes to organize code imported from C#. We will see that later on.

### 3.4.4 Inline code

Sometimes it is convenient to insert small fragments of C# or OCaml code into the code generated by CSML. CSML lets one use the following syntax in C# or

OCaml parts of CSML scripts:

```
inline [* ... *]
```

For instance, the following CSML script defines an additional OCaml function defined in terms of those imported by CSML:

```
mlfile "mymodule.ml"

module M: sig
  val f_with_id: int -> string -> unit = static Foo.f
  val f_without_id: string -> unit = static Foo.f
  inline [*
    let f ?id s =
      match id with Some id -> f_with_id id s | None -> f_without_id s
  *]
end
```

The same syntax can be used to insert comments used by tools that generate documentation from source code:

```
csfile "myfile1.cs"

namespace Foo {
  inline [*
    /// <summary>
    /// This is a very important class.
    /// </summary>
  *]
  public class A {
  }
}
```

## 3.5 Exceptions

Exceptions are properly supported by CSML. An exception raised in one language is wrapped as an exception of the other language and it can be captured there. If it goes back to its native language, then the original exception value is extracted (as opposed to being wrapped again).

| C# | OCaml |
|---|---|
| System.Exception | exn |

The OCaml module `Csml_iface`, which is part of CSML's runtime library defines an OCaml exception that wraps all the C# exceptions:

```
exception Csharp_exception of string * string * cshandle
```

The first argument is the C# exception's type name. The second argument is the exception's message. The third one is an opaque pointer to the original exception.

Similarly, the CSML runtime library defines a C# exception class `LexiFi.Interop.MLException` that encapsulates OCaml exceptions. The `Message` property of this exception class calls the OCaml function `Csml_iface.print_exception` (of type `exn -> string`). It is possible to provide a custom implementation for this function, using the `Csml_iface.print_exception_ref` reference. The default implementation uses `Printexc.to_string` and can be extended by the OCaml part of the application with custom printers for some exceptions (they can be registered with `Csml_iface.register_exception_print`).

## 3.6 Exporting OCaml values opaquely

Up to this point, we have only seen how to import functions (from OCaml to C#) and static method (from C# to OCaml) that operate on built-in types (basic types or structural types). It is possible to extend to set of types that can flow from one language to the other. In this section, we will see how to create C# classes that wraps OCaml values in an opaque way. By opaque, we mean that the values themselves always stay in their native heap (here the OCaml heap); they are never copied. Instead, a pointer from the target language to the native one is used and wrapped as a typed object.

### 3.6.1 Running example

Let us consider the following OCaml unit `opaque_binding.ml`.

```
type t = {
    mutable foo: int;
    mutable bar: int;
  }

let to_string c = Printf.sprintf "foo = %i, bar = %i" c.foo c.bar
let get_foo c = c.foo
let set_foo c x =  c.foo <- x
let get_bar c = c.bar
let set_bar c x = c.bar <- x
let create foo bar = { foo = foo; bar = bar }
let version () = "1.0"
```

This module defines a type `t` that we would like to bind to a C# class. Since values of type `t` encapsulate mutable states, it would be inappropriate to copy their content to C#.

The following section is enough to let CSML knows about the type `t` and produces the needed machinery to map values of this type to and from instances of a class `Counter`:

```
public class Counter = Opaque_binding.t {
}
```

With this declaration, it is now possible to import functions that operate on type `Opaque_binding.t` as in:

```
public static class MyClass {
  public static void SetFoo(Counter, int) = Opaque_bindng.set_foo;
}
```

The static method `SetFoo` wraps the function `Opaque_binding.set_foo`, which must have type `Opaque_binding.t -> int -> unit` (because the OCaml type `Opaque_binding.tw` is in correspondence with the C# class `Counter`).

It would make more sense to import functions directly related to the type `Opaque_binding.set_foo` as components of the class `Counter`. Indeed, CSML lets us define constructors, instance methods and properties for C# class that wraps OCaml types opaquely. Constructors can be defined from functions that return values of the wrapped type. Instance methods reflect functions that take the wrapped value as an implicit extra first argument. A property getter takes the wrapped value as their only argument and return the value of the property. A property setter takes two arguments: the wrapped value and the new value for the property.

It is possibly to define overloaded constructors and methods following the regular C# mechanism.

```
public class Counter = Opaque_binding.t {
  public Counter(int, int) = Opaque_binding.create;
  public Counter() = [* fun () -> Opaque_binding.create 0 0 *];
  public static Counter Create(int, int) = Opaque_binding.create;
  public int Linear(int) =
   [* fun c i ->
        c.Opaque_binding.foo + i * c.Opaque_binding.bar *];
  public int Bar {
     get = Opaque_binding.get_bar;
     set = Opaque_binding.set_bar;
  }
  public int Foo {
```

```
    get = [* fun c -> c.Opaque_binding.foo *];
    set = [* fun c x -> c.Opaque_binding.foo <- x *];
  }
  public static string Version { get = Opaque_binding.version; }
  public override string ToString() = Opaque_binding.to_string;
}
```

This is our first use of the `[* ... *]` notation. It is used here to put inline OCaml definitions for C# components. All such pieces of OCaml code are put in the OCaml stub file (see Section 3.4.1), so the code above works as long as we do not add an explicit interface `opaque_binding.mli` to hide the concrete definition of the type `t`.

The method `ToString` overrides the corresponding method from the parent class `Object`.

### 3.6.2 Extending the class with custom code

It is not currently possible to specify in the CSML script that a C# class inherits from another class or implements a C# interface, but it is possible to add this information in a hand-written C# file, using the fact that the classed generated by the CSML compiler are partial. Here is an example of such a hand-written file to show how to express that `Counter` implements some interface:

```
public interface ICounter {
  int Linear(int i);
}

public partial class Counter : ICounter {
}
```

Of course, it is also possible to add arbitrary extra components to the class `Counter`.

### 3.6.3 About garbage collection

When an OCaml value must be mapped opaquely to C#, a global root is registered with the OCaml garbage collector and an handle to it is passed to C#. This is to ensure that the OCaml value will not be discarded by the garbage collector while the C# code still has a pointer to it. The global root is released when the C# object that wraps this handle is released by the C# garbage collector.

It is possible to release explicitly the global root hidden behind the C# wrapper. To do this, one must declare a special method like that:

```
public class Counter = Opaque_binding.t {
...
  public void KillMe() = kill;
...
}
```

The keyword `kill` is recognized by the CSML compiler. When the method `KillMe` is called, the underlying handle is released. Any further access will result in an exception being raised.

## 3.7 Exporting C# values opaquely

In the previous section, we have seen how to wrap OCaml values as C# objects in a opaque way. We can do the same the other way around, that is, we can bind arbitrary C# types to custom abstract OCaml types. At runtime, C# values are kept in the C# heap and the OCaml program only manipulates pointers to C# values.

### 3.7.1 Running example

Let us consider the following C# class that we want to bind to OCaml.

```
public class MyClass {
  private int i = 0;
  public MyClass() { }
  public MyClass(int i) { this.i = i; }
  public int Value { get { return i; }   set { i = value; } }
  public void Bump() { i++; }
  public void Bump(int x) { i += x; }

  private static MyClass glb = null;
  public static Global { get { return glb; }  set { glb = value; } }
}
```

We can use the following CSML script to bind this class to an abstract type and import its components as OCaml functions.

```
type t = MyClass

val create: unit -> t = ctor
val create_init: int -> t = ctor
val get: t -> int = get Value
val set: t -> int -> unit = set Value
val bump: t -> unit = instance Bump
```

```
val bump_n: t -> int -> unit = instance Bump
val global: unit -> t = static get MyClass.Global
```

As we can see, we describe on the right-hand side of each declaration (after the `=` sign) what C# component should be imported. We had seen previously how to import C# static method (`static` keyword) and static properties (`static get`, `static set`). Here we see examples of how to import constructors (`ctor` keyword), getter and setters for instance properties (`get`, `set`) and instance methods (`instance`). Note that for non-static components, the right-hand sides do not mention the class to which the components belong. This class is defined implicitly by the first argument (for properties and instance methods) or by the result type (for constructors).

Note that the `get` and `set` keywords can be used to access not only public properties, but also public fields.

### 3.7.2 Special methods

The CSML compiler recognizes other possible keywords on the right-hand side of declarations in OCaml parts. Here we consider three special pseudo-methods.

**Checking for nullness** The `MyClass.Global` property can return `null`. As a consequence, the function `global` above might return something which is not really an object. It is possible to define a function that checks whether a value of type `t` is actually `null` or not:

```
val isitnull: t -> bool = isnull
```

**Casting** Imagine we have defined two abstract types `t1` and `t2` to wraps two C# classes `C1` and `C2`. We can define a function that cast objects of class `C1` to class `C2`:

```
val t2_of_t1: t1 -> t2 = cast
```

The C# code generated for this cast uses the C# `as` operator. As a consequence, this function can return null. If we want to get an option instead of a possibly `null` value (which can be checked with an `isnull` function as above), we can use the `nullable` pseudo-type (See Section 3.3.2):

```
val t2_of_t1: t1 -> t2 nullable = cast
```

**Killing the pointer** One can define a function that kills the reference to the underlying C# object. If such a function is not called on a value that wraps a C# object, one must wait for the OCaml value to be released by the OCaml garbage collector before the reference is released (and the C# object can potentially be released by the C# garbage collector). This is done by the special keyword `kill`:

```
val killit: t1 -> unit = kill
```

### 3.7.3 Indexed accesors

It is possible to define OCaml functions to access indexed properties. For instance, if the C# class bound to the type `t` has a property `p` with an indexer that maps strings to integers, it is possible to define OCaml function like that:

```
val get_p: t -> string -> int = indexed get p
val set_p: t -> string -> int -> unit = indexed set p
```

If the class bound to `t` itself has such an indexer, we can use `this` instead of the property name:

```
val get_p: t -> string -> int = indexed get this
val set_p: t -> string -> int -> unit = indexed set this
```

There are also static variants for the indexed accessors. Imagine that the C# expression `Foo.Instance` resolves to an object with a property `p` that defines an accessor as above. We can define:

```
val get_p: string -> int = static indexed get Foo.Instance.p
val set_p: string -> int -> unit = static indexed set Foo.Instance.p
```

### 3.7.4 Accessing sub-components

It is possible to use the `static`, `instance` and other keywords to bind directly sub-components as long as they are expressible as a qualified identifier (with the dot notation). For instance, we could create OCaml functions that access directly components of the `Global` property:

```
val bump_global: unit -> unit = static MyClass.Global.Bump
val get_global: unit -> int = static get MyClass.Global.Value
```

### 3.7.5 Ignoring the result

If we want to import a C# method as an OCaml function, we need to provide the result type of the method, even if we do not care about the result. For instance, if the method takes an integer and returns a string, we must write something like:

```
val f: int -> string = static Foo.f
```

It it possible to ignore the result explicitly so that we can use `unit` as the return type:

```
val f: int -> unit = ignore static Foo.f
```

### 3.7.6 Weak references

In many cases, it is not necessary to keep a strong reference to the C# object. CSML makes it possible to keep only a weak reference which does not prevent the C# garbage collector from reclaiming the underlying C# object. This is especially interesting if we have cycles between the two heaps (e.g. an OCaml record that keep a pointer to a C# object, which itself keeps a pointer to the OCaml record), because in that case, without weak references, we would need to explicitly break cycles (e.g. using pseudo `kill` functions and methods).

We can annotate types of values that flow opaquely from C# to OCaml with an annotation in order to produce weak references instead of strong ones. The syntax is $\alpha$ `weak` in OCaml parts and `Weak<T>` in C# parts.

```
val global_weak: unit -> t weak = static get MyClass.Global
...

public class M {
  public static void F(Weak<MyClass>) = Module.f;
}
```

## 3.8 Binding C# classes to OCaml classes

We have seen that arbitrary C# objects can be wrapped as opaque pointers in OCaml. It is also possible to wrap them as OCaml objects.

We recall here an example seen above:

```
type t = MyClass
```

```
val create: unit -> t = ctor
val create_init: int -> t = ctor
val get: t -> int = get Value
val set: t -> int -> unit = set Value
val bump: t -> unit = instance Bump
val bump_n: t -> int -> unit = instance Bump
val global: unit -> t = static get MyClass.Global
```

We replace the first line with:

```
class t = MyClass : object
end
```

The effect of this declaration is that C# objects class `MyClass` will be reflected in OCaml as objects. The other definitions are still valid, but the ones that take `t` as their first argument can be replaced by methods:

```
class t = MyClass : object
  method get: int = get Value
  method set: int -> unit = set Value
  method bump: unit = instance Bump
  method bump_n: int -> unit = instance Bump
end
val create: unit -> t = ctor
val create_init: int -> t = ctor
val global: unit -> t = static get MyClass.Global
```

Note that the OCaml constructor cannot be used to reflect the C# constructors. (It is used internally by CSML.)

It is possible to define OCaml methods as importing *static* C# methods, as long as these C# methods takes an object of the correct class as their first argument.

In addition to the method described explicitly in the CSML script, the generated class also contains some internal methods.

### 3.8.1 Inheritance

Imagine that `C2` is a subclass of `C1` in C#. It is possible to reflect this in the imported classes:

```
class t1 = C1 : object
  ..
end
```

```
class t2 = C2 : object
  inherit t1
  ...
end
```

The OCaml type `t2` will then be a subtype of `t1`. Up-casting can thus be implemented simply with the OCaml syntax `(e :> t1)`.

For down-casting, we can use, as expected, the `cast` keyword. We can simply define a function to do the cast:

```
val t2_from_t1: t1 -> t2 nullable = cast
```

Or we can add this as a method of `t1`:

```
class t1 = C1 : object
  val to_t2: t2 nullable = cast
end
```

## 3.9 Structurally reflecting OCaml datastructures into C# classes

We have seen how to manipulate opaque handle on arbitrary OCaml values from C#. Sometimes, we would like to transform structured OCaml values into native C# values. CSML is able to produce C# class declarations to reflect OCaml variant and record types, and to translate the OCaml value to and from this C# representation.

### 3.9.1 Records

Imagine that we want to bind some record type defined in OCaml:

```
type myrecord = {
  x: int;
  y: myrecord option;
}
```

As we have seen before, it is possible to reflect this type opaquely to C# with a CSML script like:

```
public class MyRecord = Structured_mapping.myrecord
{
}
```

If we want instead CSML to copy the record fields into a real C# object, we need to inform CSML about the concrete definition of the type `Structured_mapping.myrecord`. We do this by repeating the definition of the type (with fully explicit paths for OCaml types):

```
public class MyRecord = Structured_mapping.myrecord =
  {
    x: int;
    y: Structured_mapping.myrecord option;
  }
{
}
```

From this script, CSML produces a class declaration which implements the following signature (plus some additional fields for its own internal use):

```
public partial class MyRecord {
  public MyRecord(int, LexiFi.Interop.Option<MyRecord>);
  public int x { get; set; }
  public LexiFi.Interop.Option<MyRecord> y { get; set; }
  ...
}
```

Also, the types `Structured_mapping.myrecord` and `MyRecord` are now available in the rest of the CSML script: values of those types are copied structurally when crossing the boundary between the languages.

We can see that the class defines one constructor that takes one argument for each field of the record, and one property (with a getter and a setter) for each field.

**Custom field names** CSML allows one to specify custom names for fields. For instamce, if we want the OCaml field `x` to produce a property named to `First` in C#, then we can write:

```
public class MyRecord = Structured_mapping.myrecord =
  {
    x as First: int;
    y: Structured_mapping.myrecord option;
  }
{
}
```

**Private flag** If the OCaml type were declared private, then we should add the same flag to the CSML script:

```
public class MyRecord = Structured_mapping.myrecord =
  private
  {
    x: int;
    y: Structured_mapping.myrecord option;
  }
{
}
```

### 3.9.2 Variants

While OCaml record types have a natural translation into C#, this is less true for variant types. CSML suggests one possible encoding of variant types into C#. Let us consider the following OCaml type declaration:

```
type myvariant =
  | A
  | B of string * int
  | C of myvariant
```

As for records, the CSML script to import this type to C# needs to repeat the type definition:

```
public class MyVariant = Structured_mapping.myvariant =
  | A
  | B of string * int
  | C of Structured_mapping.myvariant
{
}
```

The CSML compilers produces an abstract class `MyVariant`:

```
public abstract partial class MyVariant {
  public partial class A : MyVariant {
    public A();
  }
  public partial class B : MyVariant {
    public B(string, int)
    public string TVal0 { get; set; }
    public int TVal1 { get; set; }
```

```
  }
  public partial class C : MyVariant {
    public C(MyVariant)
    private MyVariant val0;
    public MyVariant TVal0 { get; set; }
  }
  public abstract class MatchVoid {
    abstract public void A();
    abstract public void B(string,int);
    abstract public void C(MyVariant);
    public void run(MyVariant x);
    public static void RunMatch(MyVariant,
                                LexiFi.Interop.ArrowVoid,
                                LexiFi.Interop.ArrowVoid<string,int>,
                                LexiFi.Interop.ArrowVoid<MyVariant>);
  }
  public abstract class Match<T> {
    abstract public T A();
    abstract public T B(string,int);
    abstract public T C(MyVariant);
    public T run(MyVariant);
    public static T RunMatch(MyVariant,
                             LexiFi.Interop.Arrow<T>,
                             LexiFi.Interop.Arrow<string,int,T>,
                             LexiFi.Interop.Arrow<MyVariant,T>);
  }
```

There is one inner sub-class for each OCaml constructor of the variant type. Each one looks a lot like the class generated for a record type, with one constructor and properties for the arguments.

**Pattern matching** The inner classes `MatchVoid` and `Match<T>` let us define pattern matching on the type `MyVariant` (`Match<T>` is used when the pattern matching returns a value). The methods `RunMatch` can be called directly; they must be given the object of type `MyVariant` to be inspected and one delegate for each possible case. The other possible way to implement pattern matching is to sub-class `MyVariant.MatchVoid` or `MyVariant.Match<T>` (for some specific `T`). The sub-class must implement one method for each OCaml constructor of the variant type. This ensures that the matching is exhaustive. Here is an example of such a pattern matching class:

```
public class VariantMatching : MyVariant.Match<int> {
  override public int A() { return 1; }
  override public int B(string s, int i) { return i; }
  override public int C(MyVariant x) { return this.run(x); }
```

```
  public static int Match(MyVariant x) {
    return (new VariantMatching()).run(x);
  }
}
```

**Customizing the generated class** There is one concrete sub-class on the generated class for each case of the original OCaml type. By default, their names correspond to the OCaml constructor, but it is possible to provide a custom name. For instance, if we want to have a class `MyVariant.Bee` instead of `MyVariant.B`, we can do:

```
public class MyVariant = Structured_mapping.myvariant =
  | A
  | B as Bee of string * int
  | C of Structured_mapping.myvariant
{
}
```

It is also possible to give custom names for the arguments of each constructors, instead of the default `TVal0`, `TVal1`, ... To do this, we use a record-like notation:

```
public class MyVariant = Structured_mapping.myvariant =
  | A
  | B as Bee of { name: string; amount: int }
  | C of Structured_mapping.myvariant
{
}
```

**Private flag** As for records, tf the OCaml type were declared private, then we should add the same flag to the CSML script.

## 3.10 Mapping C# enumerations into OCaml variants of polymorphic variants

The CSML compiler supports binding C# enumerations to OCaml variants. This feature is experimental and is likely to change. It is currently not documented.

**TODO:**...

## 3.11 Resolution of recursive types

We have seen in the previous sections how to extend the set of admissible types by specifying bindings between OCaml and C# types. As expected for C# looking code, it is possible to refer in a C# section to any type defined anywhere in the script. For instance, the following is legal:

```
csfile "myfile1.cs"

namespace X {
  class A {
    public static void f(B, C) = Foo.f;
    // B refers to the one defined in myfile2.cs, which reflects an ML type
    // C refers to the class C to be provided by an extra C# source file
    //   of the project
  }
}

csfile "myfile2.cs"

class B = Foo.t { }

mlfile "mymodule1.ml"

type s = C
  // This informs csml about the C# type C
```

In this example, the module `foo.ml` must define a type `t` and a function `f` of type `t -> Mymodule1.s -> unit`.

Maybe more suprisingly, the type definitions are also fully mutually recursive for the OCaml sections. The following is thus valid:

```
mlfile "mymodule1.ml"

module A: sig
  type t = Foo.A
  val a_to_b: t -> B.t = cast
end

module B: sig
  type t = Foo.B
  val b_to_a: t -> A.t = cast
end
```

To support this kind of recursion, CSML actually generates all OCaml type declarations together (with mangled names) and then use those types. The code above would produce a file `mymodule1.ml` that looks like:

```
type mymodule1_a_t
type mymodule1_b_t

module A = struct
  type t = mymodule1_a_t
  let a_to_b : mymodule1_a_t -> mymodule1_b_t = ...
end

module B = struct
  type t = mymodule1_b_t
  let b_to_a : mymodule1_b_t -> mymodule1_a_t = ...
end
```

To support recursion between several units, if the CSML script has several OCaml sections, then all the type declarations go at the beginning of the file generated from the first one (which thus has a special role). In general this is ok. But if this file comes with an explicit interface (`.mli`), then those mangled type declarations might not be available where they needed. In that case, a solution is to add an extra empty OCaml section at the beginning of the CSML script to act as a container for the type declaration. For instance, the following script will force all the type declaration to go to a specific file `mymodule_types.ml`.

```
mlfile "mymodule_types.ml"


mlfile "mymodule1.ml"

module A: sig
  type t = Foo.A
  val a_to_b: t -> B.t = cast
end

module B: sig
  type t = Foo.B
  val b_to_a: t -> A.t = cast
end
```

## 3.12 Dependent CSML scripts

It is of course possible to use several CSML scripts for a single application. This modularity is especially useful if we want to use CSML to import library from one language to the other.

As we have seen, in addition to specifying how to import components from one language to the other (functions, methods), a CSML script also defines a correspondence between C# and OCaml types. In a CSML script, it is possible to refer to an external script with a `use` directive:

```
mlstub ...
csstub ...

use "external.csml"
```

Now, in the rest of this script, it is possible to use the type correspondence as defined by `external.csml` (and the ones defined in scripts referred to from this one, and so recursively). For instance, if `external.csml` defines that an OCaml type `A.t` is an opaque counterpart for the C# type `Foo.A`, then we can use those OCaml and C# types in the current script.

# Chapter 4

# Using CSML

## 4.1 Using the compiler

The CSML compiler is invoked with a simple command line like:

```
csml myscript.csml
```

where `myscript.csml` is a CSML script. The compiler reads the script (and any script referenced from it, see 3.12) and produces all the files according to the `mlstub`, `cstub`, `mlfile` and `csfile` declarations.

It is important to note that the compiler does not access any other file. In particular, it never parse OCaml `.cmi` files and it does not use .Net reflection to check the type of the imported components. Instead, it produces OCaml and C# source files that incorporate static type checking. If the C# or OCaml compiler fail on one of these files, it means that the CSML script contains an invalid type for some of the imported component (usually, the message issued by the compiler is clear enough to indicate the problem).

## 4.2 Computing dependencies

The following command line asks the CSML compiler to produce on its standard output a description of the dependencies implied by the CSML script, in a format suitable for inclusion in a Makefile:

```
csml -dep myscript.csml
```

## 4.3 Linking

A typical mixed C# and OCaml project is made of a number of C# and OCaml units. Some of them are produced by the CSML compiler and some of them are

provided explicitly. There is some freedom in the way all these units are linked together.

### 4.3.1 Initialization

The final application is always a .Net assembly that contains some C# parts of the application (some of them can be linked in external DLLs). It must start by calling all the initialization methods for the CSML scripts that are part of the application. For instance, if one of those script has a directive like:

```
csstub "cs_stub.cs" InitClass
```

then the main program must call `Lexifi.Interop.InitClass.Init();`.

### 4.3.2 Static linking

In this linking strategy, the OCaml part of the application is linked into a DLL with the `-output-obj` option of `ocamlc` or `ocamlopt`. You must include two libraries `csml_standalone.cma` and `csml_init.cma` into this DLL in addition to all the libraries and user code needed by the application. Note that this DLL contains not only the code for the application but also the OCaml runtime system.

The C# part of the application is linked as a .Net assembly by the C# compiler. In addition to the code of application, you must link the C# file `csml.cs` into this assembly. It is also necessary to link a small C# file that indicates the name of the DLL produced above. The CSML compiler can generates such a file:

```
csml -dllbind mydll.dll > mydll_ptr.cs
```

where `mydll.dll` is the name of the DLL with the OCaml application and runtime system, and `mydll_ptr.cs` is the name of the C# file to be created. This file must then be linked with `csml.cs` and the rest of the application.

### 4.3.3 Dynamic linking

The CSML distribution comes with two pre-linked DLLs `csml_ml_byt.dll` and `csml_ml_opt.dll` that contain the CSML runtime library and the OCaml runtime (in bytecode or native form). These DLLs also contains the following OCaml libraries: `dynlink`, `bigarray`, `unix`. The distribution also comes with two files `csml_ml_byt.cs` and `csml_ml_opt.cs` that correspond of the output of `csml -dllbin` applied to these DLLs.

To use these DLLs, one must link the application into one or several files that can be linked (`.cmo` or `.cma` in bytecode, `.cmxs` in native code). The C# part of the application can use the built-in static method `LexiFi.Interop.Csml.LoadFile`

to load such files dynamically (this must be done after the call to the initialization methods in C#). This method automatically translate the `.cmo` and `.cma` file extensions to `.cmxs` when the underlying OCaml runtime system is the native one.

When one uses the default DLLs, one can also use `csml_byt.dll` or `csml_opt.dll`. They are .Net DLL that contain the CSML C# runtime (`csml.cs`) and references to the default DLLs (i.e. `csml_ml_byt.cs` and `csml_ml_opt.cs`).

### 4.3.4 Linking the OCaml code without the C# part

In some cases, it is useful to link the OCaml modules produced by the CSML compiler even when the .Net runtime is not available in the current application. In this case, calling an OCaml function that imports a C# component will raise an exception (actually, it stops the application currently). To link the OCaml modules produced by CSML, one must include `csml_standalone.cma` and not `csml_init.cma` (resp. `.cmxa` in native code).

It is possible to check whether the C# runtime is available with the Boolean `Csml_iface.csharp_available`. It is also possible to check for each individual function imported from C# to OCaml whether it is available in the current program. Indeed, for each such function `M.f`, CSML also declares a function `M.f___available` (i.e. the name of the function followed by three underscores and then `available`) of type `unit -> bool` which allows on to check whether `M.f` is available.

## 4.4 A note on initialization order

Currently, OCaml modules that are part of the application are not allowed to call C# methods in their initialization code. Indeed, the OCaml runtime is started before the C# components are exported to OCaml. This might change in the future.

## 4.5 A note on environment variables

It is important to know that Windows provide two APIs to access environment variables of the current process. Actually, the C runtime library keeps its own copy of the environment, which is the one used by the OCaml functions. This is problematic because the .Net framework uses the other set of functions. As a consequence, the two environment might become desynchronized: a change done in C# will not be visible in OCaml. To alleviate this problem, the initialization part of the CSML OCaml runtime library copy the .Net environment to the OCaml one. As a consequence, any change to the environment made before this initialization is reflected on the OCaml side (this allow to pass some information from C# to OCaml).

# Chapter 5

# Formal syntax

This chapter gives a formal description of the grammar for CSML scripts.
**TODO:**...